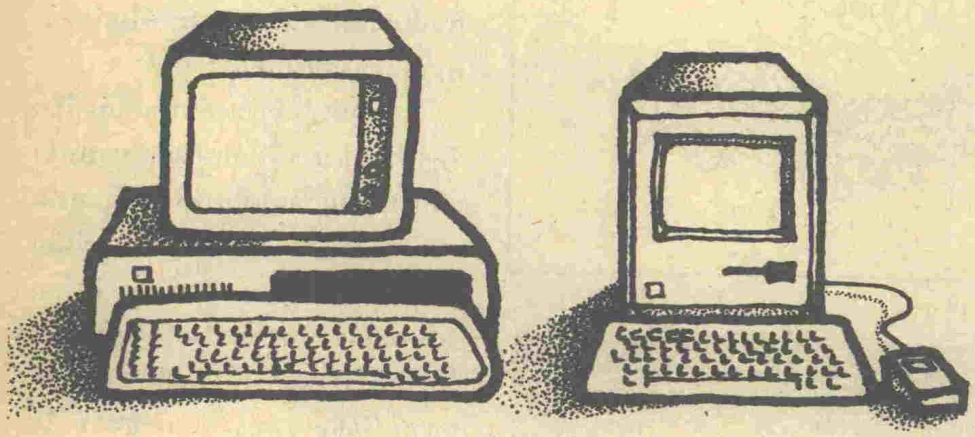
เรื่องน่าอ่าน
ดร.ปัญญา เปรมปรีดิ์

Monolithic VS Distributed



เปรียบเทียบข้อดีและข้อเสียของ 2 ระบบโปรแกรม เพื่อใช้เป็นแนวทางในการพัฒนาต่อไป

มีคนจำนวนหนึ่งที่ชอบออกแบบระบบให้เป็นโปรแกรมใหญ่เพียงโปรแกรมเดียว แล้วก็มีคนอีกจำนวนหนึ่งที่ชอบแยกงานออกเป็นโปรแกรมหลายโปรแกรม แนวคิดของกลุ่มแรกนั้นเรียกว่า "Monolithic Design" ส่วนแนวคิดของกลุ่มที่ 2 นั้นเรียกว่า "Distributed Design" แนวคิดของสองกลุ่มนี้ขัดแย้งกันอย่างแรง และบางครั้งบางคราวมันมีผลถึงขนาดทำให้ทำงานร่วมกันไม่ได้ แต่เรื่องนี้เราก็ไม่เคยเอามาพิจารณาเปรียบเทียบกันอย่างจริง ๆ จัง ๆ เราจึงต่างคนต่างคิด และต่างคนต่างใช้ ซึ่งทำให้เกิดปัญหาได้ เช่น ประสิทธิภาพไม่ดี (ข้างไหนยังไม่รู้) ดูแลบำรุงรักษาข้ามกลุ่มไม่ได้ สร้างแรงกดดันให้กับผู้ทำซิสเต็มส์ซอฟต์แวร์สร้างความวุ่นวายและสับสนขึ้นในหมู่ยูสเซอร์ วันนี้เรามาดูข้อดีข้อเสียของ 2 แนวคิดนี้กันดีไหม?

กลุ่มโมโนลิทิค

ข้อดีข้อที่ 1

ข้อดีข้อแรกของกลุ่มโมโนลิทิคก็คือ ยูสเซอร์จำได้ง่าย มันเป็นโปรแกรมเพียงโปรแกรมเดียว เช่น LOTUS หรือ WSTAR เมื่อคิดจะใช้ก็ไปเรียกโปรแกรมนั้นขึ้นมา เขามักจะออกแบบให้เป็นเมนู ผู้ใช้สามารถเลือกว่าจะทำเรื่องอะไร เมนูจะเปลี่ยนไปตามฟังก์ชันที่ต้องการ เลือกลงไปจนถึงจุดที่จะเอาข้อมูลเข้า หรือใช้งานแฟ้มข้อมูลเมื่อเสร็จแล้วก็ถอยกลับ (Exit) เป็นขั้น ๆ และกลับไปที่เมนเมนูได้ การออกแบบระบบแบบนี้มองดูว่าเป็นเรื่องพื้น ๆ แต่ผลทางด้านการยอมรับของยูสเซอร์นั้นมีสูงมาก ยูสเซอร์จะมีความเชื่อมั่นว่าจะใช้งานระบบนี้ได้ เรื่องนี้ถ้าเปรียบเทียบกับงานแบตช์โปรเซสซิงที่เราเคยใช้จะพบว่า งานแบตช์โปรเซสซิงนั้นเป็นงานที่ยูสเซอร์ใช้เองไม่ได้ 1 ระบบมักจะประกอบไปด้วยโปรแกรม 40-50 โปรแกรม แต่ละโปรแกรมนั้นยังต้องมีการเซตทางเลือก (Option) และต้องจัดให้โปรแกรมเหล่านี้วิ่งต่อเนื่องกันเป็นจ๊อป (Job) ระบบแบตช์โปรเซสซิงต้องให้คนในศูนย์คอมพิวเตอร์เป็นคนจัดเข้าเครื่องและถึงแม้จะอยู่กับงานนั้นมาเป็นปี ๆ ก็ยังทำงานผิดพลาดบกพร่องได้ ระบบโมโนลิทิคจึงดูจะดีกว่า

ข้อดีข้อที่ 2

ข้อดีข้อที่ 2 ของโปรแกรมประเภทโมโนลิทิคก็คือ ตัวแปรต่าง ๆ ที่อยู่ในโปรแกรมนั้นจะถูกเรียกใช้เมื่อไหร่ก็ได้

ดังนั้นขั้นตอนการทำงานภายในโปรแกรมจะรวดเร็วกว่า มันไม่ต้องมาเสียเวลาในการคำนวณหาใหม่ ถ้าเป็นแบบแยกเป็นหลายโปรแกรม และวิ่งเป็นขั้นตอนแบบแบตช์โปรเซสซิงเราจะพบว่า เมื่อจบโปรแกรมที่ 1 ข้อมูลหรือตัวแปรทุกตัวถูกทิ้งไปหมด นี่เป็นการสูญเสียอย่างใหญ่หลวง แต่อย่างไรก็ตาม คนที่เขียนโปรแกรมแบบชนิดเรียกใช้ตัวแปรที่จุดไหนก็ได้นั้น จะมีปัญหาเรื่องดีบักคือเขียนโปรแกรมผิดได้ง่าย แล้วค้นหาที่ผิดได้ยาก คนส่วนใหญ่ใช้วิธีสร้างเป็นซับรูทีน (Sub-routine หรือ Procedure) แล้วในโปรแกรมนั้นจะกระโดดไปใช้เป็นระยะ ๆ แต่ละครั้งที่กระโดดกลับก็จะสูญเสียข้อมูลและตัวแปรต่าง ๆ ที่คำนวณไว้แล้วเช่นกัน แต่เขียนโปรแกรมแบบใช้ซับรูทีนกลับเขียนได้ง่ายกว่า สั้นกว่า และโอกาสผิดพลาดมีน้อยลง การสร้างเป็นซับรูทีนนี้ก็เป็นแนวคิดของกลุ่ม Distributed นั่นเอง ดังนั้นคนที่เขียนเป็นโปรแกรมใหญ่โปรแกรมเดียวแต่อาศัยเรียกใช้ซับรูทีนหลาย ๆ ชนิดก็คือพวกที่ผสมผสานทั้ง 2 เทคนิค พวกนี้ไม่ใช่ "Pure Monolithic"

ข้อดีข้อที่ 3

ข้อดีข้อที่ 3 ของกลุ่มโมโนลิทิคก็คือ มันกินเนื้อที่น้อยมาก เช่น ทั้งหมดอาจเป็นโปรแกรมขนาด 1 เมกะไบต์เท่านั้น แต่ทำได้ทุกอย่าง ถ้าเป็นพวกแยกเป็นหลายโปรแกรม แต่ละโปรแกรมอาจเล็ก ๆ เช่น 60-150 กิโลไบต์ แต่รวมทั้งหมด 40-50 โปรแกรมก็จะกินเนื้อที่ 5-6 เมกะไบต์เข้าประเภทโมโนลิทิคนั้นบรรจุลงดิสเกตต์แผ่นเดียวได้ แต่พวกดิสทริบิ้วเต็ดจะต้องใช้ดิสเกตต์ถึงครึ่งกล่อง การติดตั้งใช้งานก็จะยุ่งยากกว่ากัน นี่ก็มีผลกระทบต่อยูสเซอร์ เพราะถ้าเป็นงานในเครื่องไมโครคอมพิวเตอร์เราก็อยากให้ยูสเซอร์ติดตั้งระบบด้วยตนเอง ซึ่งเขาพอจะทำกับโมโนลิทิคโปรแกรมได้ แต่การติดตั้งระบบที่มีหลายโปรแกรมและหลายแผ่นดิสเกตต์เขาจะรับไม่ไหว

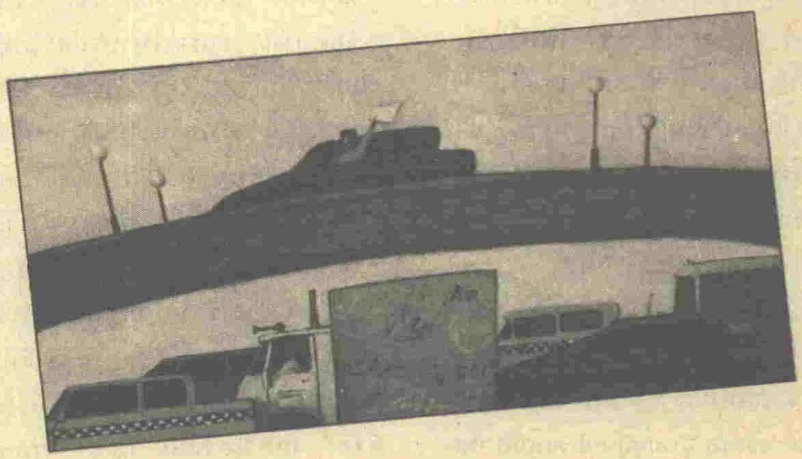
ข้อดีข้อที่ 4

ข้อดีของกลุ่มโมโนลิทิคนั้นยังมีอีกหลายข้อ แต่เพื่อให้บทความนี้ไม่เยิ่นเย้อเกินไป ผมจะขอพูดอีก 1 ข้อ คือมันพัฒนาโดยคนเพียงคนเดียว งานพัฒนาจึงเสร็จได้แน่นอน เขาจะเปลี่ยนลอจิกของโปรแกรมไปอย่างไรก็ได้ เขาไม่ต้องไปชี้แจงให้ใครฟัง การพัฒนาจึงทำได้เร็ว การทดสอบก็ทำได้ง่าย และเรามักจะพบว่าเมื่อทดสอบแล้วก็จะมีการปรับปรุงให้ดีขึ้น เขาคิดขึ้นเอง เขายินดีแก้ไขด้วยตนเอง เรื่องนี้จะแตกต่างจากพวกที่ต้องแบ่งแยกกันไปเขียนโปรแกรมหลาย ๆ คน เวลาเอาโปรแกรมมาทดสอบรวมกันก็มักจะพบกับความไม่สอดคล้องกัน มีความขัดแย้งกัน ต่างคนต่างโทษว่าเป็นความผิดของผู้อื่น โอกาสที่จะปรับแก้ให้ถูกใจยูสเซอร์ไม่มี ทั้งนี้เพราะทุกคนต่างมุ่งจะทำให้ถูกสเปคเท่านั้น

ข้อดีของกลุ่มดิสทริบิวเต็ด

ทีนี้เราจะมาดูทางด้านดิสทริบิวเต็ดกันบ้าง กลุ่มนี้จะเริ่มด้วยเหตุผลที่ว่าระบบงาน 1 ระบบงานนั้นมันมากมายและซับซ้อน เราจำเป็นต้องแยกงานออกเป็นขั้นตอนหลาย ๆ ขั้นตอน เช่น การคีย์ข้อมูลเข้า การตรวจดูความถูกต้องของข้อมูล การจัดเรียงข้อมูล การอัพเดต การดึงออกมาทำรายงาน การโยกข้อมูลที่เลิกใช้แล้วออกไปเก็บเป็นแฟ้มประวัติ แต่ละขั้นตอนนี้จะลดความซับซ้อนลง เราสามารถวางลอจิกของโปรแกรมได้โดยง่าย แต่ละโปรแกรมจะมีขนาดเล็กลง สามารถแจกจ่ายให้โปรแกรมเมอร์หลาย ๆ คนนำไปเขียนพร้อม ๆ กัน เราจึงพัฒนาได้เร็วขึ้น งานประเภทโมโนลิทิคนั้นใช้ได้กับงานเล็ก ๆ หรืองานที่มีลักษณะจำเพาะเท่านั้น ซึ่งก็ฟังดูมีเหตุผลดี ในเรื่องของการที่ต้องทิ้งข้อมูลในเมมโมรีไปเมื่อขึ้นโปรแกรมใหม่นั้นเขาก็มีข้อแก้ตัวคือ ข้อมูลส่วนใหญ่มันไม่จำเป็นต้องใช้แล้ว เราจึงทิ้งไปได้

ทั้ง 2 ค่ายต่างก็มีข้อดีข้อเสียด้วยกันทั้งคู่และว่ากันไปแล้วก็จะพบว่า การเขียนโปรแกรมนี้เป็นศิลปะ คือใครชำนาญด้านไหนก็จะทำแบบนั้นได้ดีกว่า คนชำนาญแบบดิสทริบิวเต็ดนั้นอาจเขียน



โปรแกรมทุกโปรแกรมด้วยตนเองก็ได้ ถึงแม้จะต้องเขียนหลายสิบโปรแกรม แต่เขาไม่ต้องไปอธิบายหรือชี้แจงให้ใครฟัง เขาก็อาจพัฒนาได้เร็วและถูกต้องได้ด้วย และมีอยู่เสมอ ๆ ที่ทำได้เร็วกว่ากลุ่มโมโนลิทิค

มองไปยังอนาคต

จากสถานการณ์และเหตุผลดังที่กล่าวมาข้างบนนี้ และจากการมองไปยังอนาคตข้างหน้า ทำไมเราไม่หยิบเอาข้อดีของแต่ละกลุ่มมาใช้?

เรื่องนี้เป็นไปได้ และจะต้องรีบปลูกฝังคนรุ่นใหม่ด้วย กล่าวคือ ในอนาคตเราจะมีการเชื่อมโยงเครื่องคอมพิวเตอร์เข้าหากันเป็นเครือข่าย และส่วนใหญ่จะเป็นเครือข่ายไมโครคอมพิวเตอร์ ดังนั้นมันจึงหนีไม่พ้นที่จะต้องเขียนโปรแกรมให้ทำงานผสานกัน การทำงานโดยแบ่งแยกงานแล้วกระจายไปอยู่ในซีพียูต่าง ๆ นั้นเป็นเรื่องหลีกเลี่ยงไม่ได้อย่างแน่นอน แต่งานแบบใหม่นี้เป็นชนิด "On-line" และ "Real-time" เกือบทั้งนั้น การแบ่งแยกเป็นหลาย ๆ โปรแกรมแล้วทำงานต่อเนื่องกันนั้นจะลดน้อยลง งานที่ต้องเชื่อมโยง (Interprogram Communication) และทำแบบ "On-line, Real-time" นี้จะต้องแบ่งแยกงานเป็นขั้นตอนย่อย ๆ แล้วมีการโยนต่อ ๆ กันไป ยกตัวอย่างเช่น โปรแกรมที่ 1 จะรับข้อมูลจากคีย์บอร์ดได้แล้วโยนไปให้โปรแกรมที่ 2 เพื่อตรวจดูความถูกต้อง ถ้าผ่านก็โยนต่อให้โปรแกรมที่ 3 เพื่อนำไปอัพเดตแฟ้มข้อมูล ได้ผลเป็นอย่างไรก็ส่งกลับไปให้โปรแกรมที่ 1 เพื่อโต้ตอบกับยูสเซอร์

ในการโหลดหรือเรียกโปรแกรมต่าง ๆ เหล่านี้ลงมานั้นเราควรให้มันเรียกกันโดยอัตโนมัติ ยูสเซอร์จะรู้จักแต่โปรแกรมแรกที่วิ่งอยู่เวอร์กสเตชันเท่านั้น ด้วยวิธีนี้เราจะได้ข้อดีจากทั้ง 2 กลุ่ม เวลาที่ยูสเซอร์ต้องการออกจากงานอัพเดตไปสู่การทำรายงานหรือเรียกดูข้อมูลนั้น เราอาจให้โปรแกรมแรกไปเรียกโปรแกรมประเภททำรายงานออกมาแล้วสั่งให้จบตัวเองลงไปเลย ถ้ายูสเซอร์ต้องการกลับมาสู่โปรแกรมอัพเดตอีกก็ทำได้ ทั้งนี้โดยในโปรแกรมทำรายงานนั้นจะไปเรียกโปรแกรมแรกกลับลงมาแล้วจบตัวเองไป นี่เป็นเทคนิคที่พวกเราหลายคนคุ้นเคยอยู่แล้ว ในกรณีที่เรามีเมมโมรีเหลือเฟือเราก็อาจให้โปรแกรมต่าง ๆ จบตัวเองแบบ TSR (Terminate and Stay Resident) ก็ได้ วิธีนี้จะช่วยให้การสวิตช์ไปมาระหว่างโปรแกรมทำได้รวดเร็วจนเห็นเป็นโปรแกรมเดียวกัน

ก่อนจาก

จริง ๆ แล้วมีเรื่องที่ลึกกว่านี้ เช่น การออกแบบฟังก์ชันต่าง ๆ นั้นควรทำเป็นแบบมองทั้งระบบหรือมองให้ทั่วจักรวาลไปเลย แล้วก็มาสร้างเป็นไลบรารีเอาไว้ใช้งาน ถ้าเราทำอย่างนี้ได้ งานพัฒนาระบบในอนาคตจะเป็นเรื่องที่ง่าย ไม่เคร่งเครียด ทำแล้วเชื่อถือได้ ทำแล้วเปลี่ยนแปลงได้โดยง่าย ซึ่งเป็นเป้าหมายสูงสุดของคนทางคอมพิวเตอร์เรา ผมหวังว่าบทความนี้คงพอจะช่วยจุดชนวนให้พวกเราได้คิดและได้ปรับปรุงงานกันต่อไป